On the development of IoT systems

Taivalsaari, Antero

IEEE
2018

Taivalsaari , A & Mikkonen , T J 2018 , On the development of IoT systems . in 2018 Third International Conference on Fog and Mobile Edge Computing (FMEC) . IEEE , Piscataway, NJ , pp. 13-19 , International Conference on Fog and Mobile Edge Computing , Barcelona , Spain , 23/04/2018 . https://doi.org/10.1109/FMEC.2018.8364039

http://hdl.handle.net/10138/321570
https://doi.org/10.1109/FMEC.2018.8364039

acceptedVersion

On the Development of IoT Systems

Antero Taivalsaari
Nokia Technologies, Tampere, Finland
antero.taivalsaari@nokia.com

Tommi Mikkonen
University of Helsinki, Helsinki, Finland
tommi.mikkonen@helsinki.fi

***Abstract*—A typical IoT system consists of four distinct architectural elements: devices, gateways, cloud and apps. All these elements require special skills in their development. In order to write safe, secure IoT systems, developers must be trained and experienced in four different areas of software development: embedded, cloud, web and mobile. In addition, given the distributed nature of IoT systems, distributed programming skills play a critical role. In this paper we examine the challenges in IoT system development, and summarize our observations and experiences on the necessity and co-presence of different types of software development skills in the design of IoT systems.**

I. Introduction

By now it is clear that computing environments are undergoing a major disruption. The importance of personal computers is decreasing, mobile computing has matured, and web applications and Software as a Service (SaaS) have become commonplace. The emergence of the Internet of Things (IoT) is bringing us connected devices that are an integral part of the physical world. Advances in hardware development and the availability of powerful but very inexpensive integrated chips will make it possible to embed connectivity and full-fledged virtual machines and dynamic language runtimes virtually everywhere. As a consequence, everyday things in our surroundings will become connected and programmable. The impact of this *Programmable World* disruption [1] will be every bit as significant as the mobile application revolution that was sparked when similar technological advances made it possible to open up mobile phones for third-party application developers in the early 2000s.

In this paper we reflect upon our experiences on IoT system development both from technical and educational viewpoints: what skills software developers must acquire and master when they start their journey towards IoT system and application development. These experiences are based on a variety of IoT product development efforts that we have carried out in the past four years at Nokia and Mozilla.

The key tenet in this paper is that this area is much more complex than people tend to assume. Those skills that are most prevalent among the majority of software developers today, such as familiarity with web development or mobile development for Android or iOS devices, do not suffice. Rather, IoT development projects require skills at least in four different areas of software development, reflecting the end-to-end nature of IoT systems: embedded, cloud, web and mobile software. In addition, given the fundamentally distributed nature of IoT systems, a good understanding of the key challenges in distributed systems development is a must. The faster deployment cycles that are characteristic of today's cloud-based software systems result in additional challenges and complexity as well.

II. Common End-to-End Architecture for the Internet of Things

Fundamentally, the Internet of Things is all about *turning physical objects and everyday things into digital data products and services* – bringing new value and intelligence to previously lifeless things. Effectively this means taking previously unconnected devices, connecting them to the Internet, and adding a backend service and web and/or mobile applications for viewing, analyzing and controlling those things to introduce new value and convenience. In short, an informal formula for IoT system development can be presented as follows:

Thing X + Internet + Service + Apps = Smart Thing X

Given the connected nature of smart things and the need for a backend service, IoT systems are *end-to-end* (E2E) systems that consist of a number of architectural elements that are nearly identical in all IoT systems. In our recent IEEE Software article, we pointed out that a common, generic end-to-end (E2E) architecture for IoT systems has already emerged [1] (see Fig. 1).

As depicted in Fig. 1, IoT systems generally consists of Devices, Gateways, Cloud and Applications. *Devices* are the physical hardware elements that collect sensor data and may perform actuation. *Gateways* (also sometimes known as *Hubs*) collect, preprocess and transfer sensor data from devices, and may deliver actuation requests from the cloud to devices. *Cloud* has a number of important roles, including device management, data acquisition, data storage and access, real-time and/or offline data analytics, and device actuation. *Applications* range from simple web-based data visualization dashboards to highly domain-specific web and mobile apps. Furthermore, some kind of an administrative web user interface is typically needed. Granted, IoT product offerings have their differentiating features and services as well, but the overall architecture typically follows the high-level model shown in Fig. 1.

III. Dominant Technologies in Different Parts of the End-to-End Architecture

At the surface, software development for IoT systems does not differ much from any other form of software development. When developers are working on their first IoT development

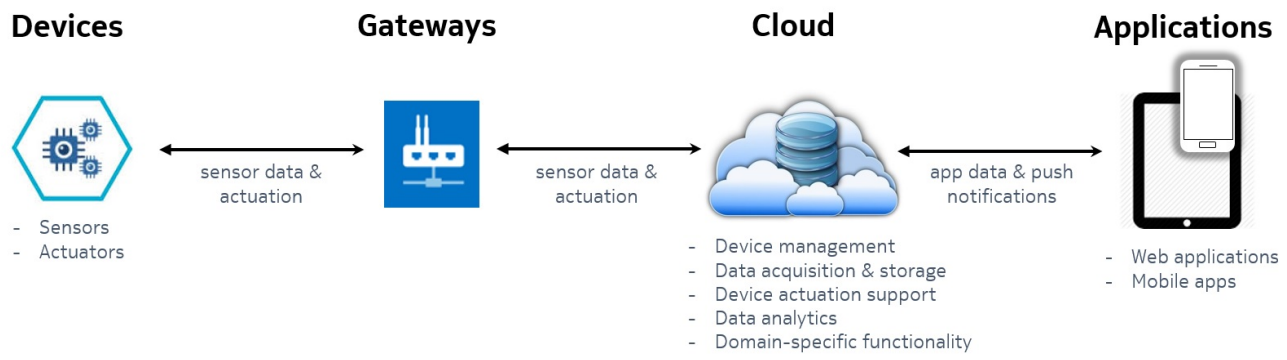

Fig. 1. Common generic end-to-end (E2E) IoT architecture.

project, they typically target a simple system that consists of a single device or board – perhaps an *Arduino* (https://www.arduino.cc/), *Tessel* (https://tessel.io/), or some version of a *Raspberry Pi* (https://www.raspberrypi.org/) – or a small, relatively homogeneous combination of such devices. Fig. 2 depicts some of today's IoT development chips and boards.

When dealing with individual devices or a handful of devices at most, IoT development does not stand significantly apart from traditional embedded or mobile software development projects. However, real-world IoT systems tend to be much more complex, consisting of hundreds or thousands or in some cases even millions or billions of IoT devices and multiple gateway solutions as well as complex cloud and analytics backends. At that level, IoT system development *is* very different from conventional web or mobile application development in which the developer is usually concerned only with a single mobile device or a single browser or PC at a time.

The majority of challenges in IoT development arise from the *distributed nature of the system* and from *intermittent, potentially unreliable connectivity and long latencies*. In complex systems, the number of IoT devices can also vary dynamically. The potentially unpredictable, highly dynamic nature of the E2E system places a lot of additional burden on the developers, as we will discuss later in Subsection IV. Furthermore, the *significantly faster deployment cycles* [2] and *DevOps* development methods [3] that are characteristic of today's cloud-based software systems result in additional complexities as well, especially if thousands of geographically distributed devices need to be updated in unison.

Before diving into those challenges, let us first take a look at each of the elemental four areas in the E2E architecture, starting from (1) Devices, and progressing via (2) Gateways to (3) Cloud and (4) Applications. Unlike many papers in the IoT area that focus on communication technologies and protocols, the focus in this paper is primarily on software technologies and software development methods.

A. *Software Technologies for* IoT Devices

The first, leftmost element in the common end-to-end IoT architecture depicted in Fig. 1 are the *Devices*. In this subsection we will summarize the software technologies and solutions used in this area.

Low-end IoT devices are driven by real-time operating systems or no operating system at all. The vast majority of today's IoT devices tend to be relatively simple. For instance, IoT devices such as lightbulbs, thermostats, remotely controlled electricity plugs, automated door locks, or air quality sensors do not commonly require complex software stacks. In order to implement a simple sensing and actuation interface, a *basic microcontroller based hardware architecture, complemented with basic drivers for sensors and actuation, will suffice*. For slightly more capable devices supporting a richer set of sensors a *real-time operating system (RTOS)* such as FreeRTOS (http://www.freertos.org/), Nucleus (https://www.mentor.com/embedded-software/nucleus/) or QNX (http://www.qnx.com/) may be required. Typical development language for low-end systems is C or C++, although even assembly code might be used in some areas.

The availability of inexpensive stock hardware is driving the industry towards "overly capable" IoT devices. In simple IoT devices, there is no need for dynamic programming support or third-party application development support in the devices themselves. Basically, all the software updates are performed by doing a *firmware update*, e.g., by reflashing the device. However, given the rapidly increasing hardware capabilities at low price points, dynamic programming capabilities are becoming increasingly feasible and common even in low-end devices. For instance, the popular *Raspberry Pi* boards (https://www.raspberrypi.org/) can provide support for a full Linux-compatible operating system at very reasonable prices. It may often be simpler and more affordable to buy such stock hardware instead of building custom HW solutions – even though stock hardware may be an overkill for the actual technical needs. For all except those IoT device solutions that require utmost attention to smallest possible size and/or lowest possible power consumption, stock hardware may offer the fastest path to success.

There are many different levels of software stacks for IoT devices. There are actually many levels of software stacks for IoT devices based on the expected programming capabili-

Fig. 2. Typical IoT development chips and boards in the mid-2010s.

ties, power budgets and underlying hardware requirements. In addition to simple "No OS" or RTOS based software stacks, there are IoT development boards that provide support for a *specific built-in language runtime or virtual machine*. For instance, the popular *Espruino* (https://www.espruino.com/) or *Tessel 2* (https://tessel.io/) IoT development boards provide built-in support for JavaScript applications, while Pycom's *WiPy* boards (https://pycom.io/development-boards) support Python development. The next level up are devices such as the aforementioned Raspberry Pi that are powerful enough to run a *full (typically Linux-based) operating system*. Compared to low-end RTOS-based or "No OS" solutions, the memory and CPU requirements (and power consumption requirements) of these more capable "Full OS" stacks are significantly higher. For instance, the desire to run a Linux-based operating system in a device bumps the minimum RAM requirements from a few tens or hundreds of kilobytes (for an RTOS-based solution) to several megabytes.

High-end wearable device platforms have software stacks that are comparable to recent mobile software platforms. At the high end of the IoT device spectrum, there are wearable device platforms such as *Android Wear* (https://www.android.com/wear/) and *Apple watchOS* (https://www.apple.com/watchos/) that are in many ways comparable to mobile software platforms from 3-5 years ago. These systems provide very rich third-party developer APIs – but also bump up the minimum hardware requirements considerably. For instance, the minimum amount of RAM required by Android Wear and Apple watchOS is half a gigabyte (512 MB) – over 10,000 times more than the few tens of kilobytes of RAM required by simple IoT sensor devices (!). Node.js (https://nodejs.org) based IoT devices are also becoming increasingly popular. For instance, the aforementioned Tessel 2 board is capable enough to run the Node.js stack, and thus serve as a standalone web server.

As can be determined from the discussion above, there is a broad range of software stacks for IoT devices, depending on the expected usage, power budget and the need to support dynamic programming and/or third-party development. The development skills required by the devices thus also vary considerably. Perhaps the most important observation here is that IoT device development is *bringing back the need for embedded, small memory software development skills* [4]. This is an interesting trend, since in the past 10-15 years many universities – at least in Europe – have scaled back their courses on embedded systems development, focusing on presumably more modern and desirable areas such as mobile software development instead. A recent Development Economics survey reports strongly confirms the demand and focus on higher-level programming skills [5].

B. *Software Technologies for* IoT Gateways

The second element in the common end-to-end IoT architecture depicted in Fig. 1 are the *Gateways*. Gateway devices have a central role in IoT systems today. The primary role of gateways is to serve as the *connectivity bridge* between IoT devices and the cloud, allowing the data collected by IoT devices to be uploaded to the cloud, and passing the actuation requests from the cloud to devices. Basically, since most IoT devices today only support near-range (local) connectivity technologies such as Bluetooth LE (https://www.bluetooth.com/specifications) or Zigbee (http://www.zigbee.org/download/standards-zigbee-specification/), the IoT devices themselves are unable to communicate with the cloud directly. Thus, an intermediary gateway solution is required for cloud connectivity.

In addition to handling cloud connectivity and data uploading, gateways may perform *preprocessing of data* and *run analytics algorithms to filter out and preselect most relevant data* before data is uploaded. They may also *generate alerts* when data values exceed certain predefined ranges. In general, since gateways typically have more computing power and other resources than IoT devices, more computing intensive functionality that needs to be carried out in the edge of the end-to-end solution is usually handled by gateways.

Today's gateway solutions can be divided broadly into two categories based on the use of the IoT system.

(1) Consumer-oriented IoT solutions typically use smartphones as gateways. IoT solutions intended for consumers often utilize the consumer's smartphone as the gateway solution. For instance, smartwatches or sports watches are usually

paired with the user's smartphone, leveraging the smartphone for data uploading and device updates. Even in those consumer-oriented solutions in which there is a dedicated gateway device – such as a "set-top home box" for controlling the user's smart home appliances – the smartphone is still used for complementing the overall end-to-end solution.

(2) Professional IoT solutions commonly use dedicated gateway devices. Professional IoT solutions tend to have special requirements that necessitate specialized gateway solutions. For instance, in industrial applications (e.g., in warehouses, factories, or mines) there may be a need for tamperproof, waterproof, dustproof and/or vibration-resistant devices, or solutions that are embedded in moving equipment such as assembly lines or forklifts.

As can be determined from the discussion above, the software technologies required for gateway development range from mobile development to embedded system development. Computational requirements of gateways are highly dependent on whether gateways are used simply for collecting and passing data onto the cloud, or whether gateways are expected to perform significant computation, e.g., by running complex analytics libraries and algorithms.

C. *Software Technologies for* IoT Cloud and Analytics

The third element in the common end-to-end IoT architecture depicted in Fig. 1 is the *Cloud*. Cloud development has evolved considerably in the past decade. Cloud computing became a hot area during the Internet boom in the late 1990s. Back in those days, developers would have to set up their own physical servers and operate their own data centers. Apart from web server and database software, pretty much all the software development had to be done from scratch.

Nowadays, nearly all the necessary implementation components are available for free as open source components. Furthermore, the availability of public cloud services such as *Amazon Web Services (AWS)* (https://aws.amazon.com/), *IBM Cloud* (https://www.ibm.com/cloud/, formerly IBM Bluemix) or *Microsoft Azure* (https://azure.microsoft.com/) has made it effortless to set up cloud environments without having to buy or own any physical server hardware.

The central elements of a typical IoT cloud backend solution are presented in Fig. 3 that is based on some of our industrial IoT development projects. Nowadays, nearly all the component areas depicted in Fig. 3 can be constructed from open source technologies. For instance, in setting up the security perimeter, developers commonly use *HAProxy* (http://www.haproxy.org/) or *NGINX* (https://nginx.org/). For data acquisition, *Apache Kafka* (https://kafka.apache.org/) is a popular solution. For data analytics, there are various solutions depending on whether the primary focus is on real-time or offline analytics; for the former area, developers commonly use *Apache Storm* (http://storm.apache.org/) or *Apache Spark* (https://spark.apache.org/), whereas offline analytics is dominated by *Apache Hadoop* (http://hadoop.apache.org/). For logging and monitoring, solutions such as *Graphite* (https://graphiteapp.org/) and *Icinga* (https://www.icinga.com/) are popular. Given the availability and maturity of open source components, the role of backend developers today could be characterized more as *software composition* or *orchestration* instead of traditional software development. In such development, the code written by the developers themselves forms only the "tip of the iceberg", while the majority of the system comes from open source code written by third parties.

Instead of building the IoT cloud solution from available open source or commercial components, it is also possible to *rent the entire IoT backend as a service*. There are popular IoT cloud services such as *Amazon AWS IoT* (https://aws.amazon.com/iot/), *Microsoft Azure IoT Hub* (https://azure.microsoft.com/en-us/services/iot-hub/) and *Nokia IMPACT IoT Platform* (https://networks.nokia.com/solutions/iot-platform) that can be used for connecting IoT devices for a recurring fee. There are also "white label" IoT cloud service providers that can set up IoT clouds for specific customers, and operate those clouds on behalf of the customers.

In general, IoT backend development has become a very popular area in recent years. As presented above, the cloud development landscape is dominated by open source technologies. According to studies, 91% of IoT developers uses open source software at least one part of their development stack [6]. The availability of open source component technologies and public clouds has led to a proliferation of IoT clouds. A recent study pointed out that there are more than 120 commercial IoT cloud solutions [7]. Given the large number of essentially identical systems, we expect significant convergence to occur in this area in the coming years.

D. *Software Technologies for* IoT Apps

The fourth central element in the common E2E IoT system architecture are the *Applications*, or *Apps* for short. By apps, we refer to the applications that are used for visualizing the data collected by IoT devices as well as for managing and controlling the devices. These apps can be divided broadly into three categories: *mobile, web and PC apps*. Mobile applications run on mobile devices such as Android or Apple iOS phones. Web applications run in a standards-compatible web browser such as Mozilla Firefox or Google Chrome. PC applications run on personal computers such as Windows, MacOS or Linux laptops.

Given that the web browser has effectively become the *de facto* execution environment for end-user software on personal computers, the development of traditional PC applications has been on the wane in recent years. Therefore, it is really only the first two categories of apps – mobile apps and web apps – that matter these days, since the majority of activities on personal computers nowadays are performed using the web browser instead of traditional installed desktop applications.

Consequently, IoT app development landscape is dominated primarily by popular mobile ecosystems – especially the Android and iOS development toolchains – as well as popular web development frameworks such as React.js or Angular. Fig. 4 illustrates the dominant software development technologies

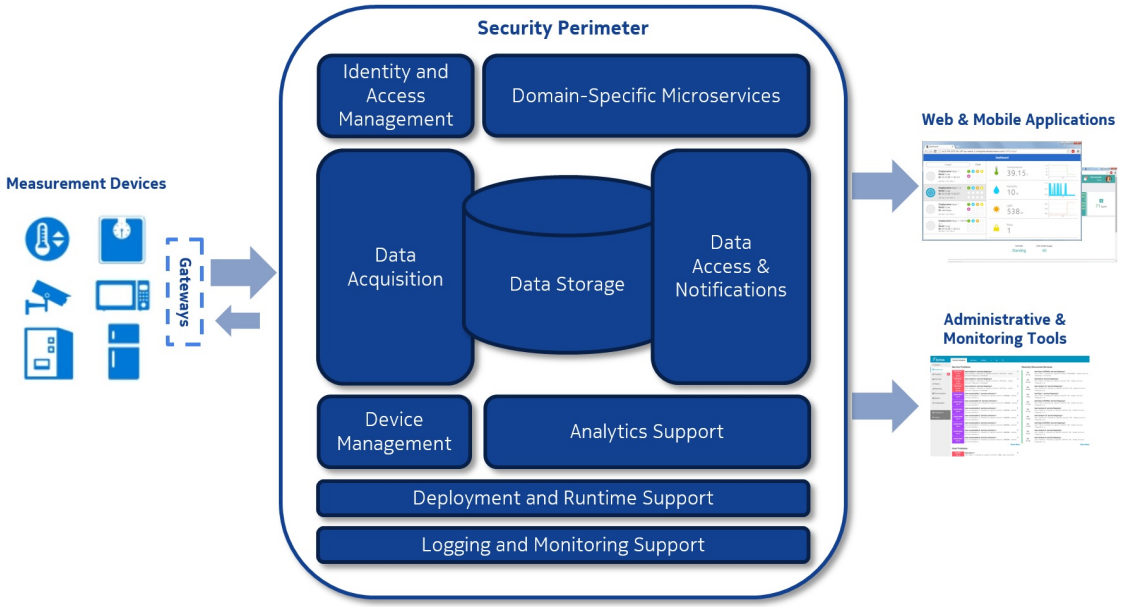

Fig. 3. Overview of a typical IoT cloud backend architecture.

required for each of the four areas in the end-to-end IoT architecture.

IV. Mastering IoT – Beyond the Expected Topics

Based on the observations presented in previous section, it is obvious that IoT development requires a broad spectrum of development technologies, languages and skills, ranging from embedded software development to cloud backend development technologies as well as mobile and web client software. For creating a complete E2E system, the development project must have people with skills in most of these areas.

Today, the vast majority of application developers have been trained to do either mobile development or web development [5]. Many of these developers tend to assume that their skills would be directly applicable to IoT development. However, this is not really true, as IoT systems have many characteristics that do not apply to mobile or web applications at all. In addition to the embedded nature of IoT devices and gateways, the distributed nature of the overall E2E system places special requirements. In the following subsections we will take a look at some areas that go above and beyond the expected topics.

A. Distributed Computing

IoT developers must consider several factors that are unfamiliar to most mobile and client-side web application developers today. Such factors include:

- multidevice programming;
- heterogeneity and diversity of devices;
- intermittent, potentially unreliable connectivity;
- the distributed, highly dynamic, and potentially migratory nature of software;
- the reactive, always-on nature of the overall system; and
- the general need to write software in a fault-tolerant and defensive manner.

In general, a typical IoT application is *continuous and reactive*. On the basis of observed sensor readings, computations get triggered (and retriggered) and eventually result in various actionable events. The programs are essentially *asynchronous, parallel, and distributed*.

The presence of these qualities may not be so obvious in the first generation IoT systems in which sensor devices are relatively simple and the majority of data processing takes place in the cloud. However, as IoT systems evolve from mere sensor data acquisition and cloud-based data analytics to comprehensive E2E systems that leverage the processing and storage capacity of the edge devices and gateways, the need for system-level thinking becomes apparent. The element of distribution is probably the single largest complicating factor in the IoT domain as the size of the overall system grows.

The *fallacies of distributed computing* are a set of assumptions that L. Peter Deutsch, James Gosling and other people at Sun Microsystems wrote down in the 1990s to summarize the assumptions that programmers will invariably make when writing software for distributed applications and systems for the first time [8], [9]:

1) The network is reliable.
2) Latency is zero.
3) Bandwidth is infinite.
4) The network is secure.
5) Topology doesn't change.
6) There is one administrator.
7) Transport cost is zero.
8) The network is homogeneous.

These assumptions commonly result (1) in the failure of the system to operate as planned, (2) a substantial reduction in system scope, and/or (3) in large unplanned expenses required to redesign the system to meet its original goals.

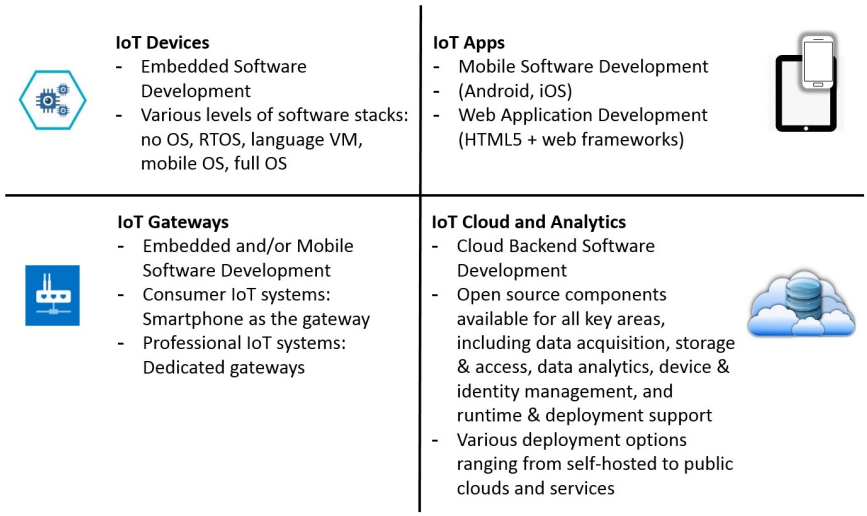

Fig. 4. Development technologies for each area in the E2E architecture.

Common examples include, e.g., applications that are written with little attention to evolving data structures or to error handling on networking errors. During a network outage, such applications may stall or infinitely wait for an answer packet, permanently consuming memory or other resources. When the failed network becomes available, those applications may also fail to retry any stalled operations or require a (manual) restart.

The hidden costs of building and maintaining software for distributed systems are almost always underestimated. According to some studies, verification and validation activities and checks amount to 75% of the total development costs for critical software [10]. As a result of the additional code needed for preparing for the fallacies of distributed computing, the actual logic of the applications gets buried under a lot of ”boilerplate” code, making programs much harder to understand and maintain. At this point, there are no good solutions for this area, except for educating IoT developers on the fallacies and pitfalls associated with distributed computing.

B. Deployment-in-the-Large and Rapid Deployment Cycles

In contrast with traditional software systems, IoT systems can consist of tens or even hundreds of thousands of independently running computing units. The large number of IoT devices and their complex topologies, interactions and different connectivity mechanisms pose interesting challenges for software development, e.g., by making it difficult to perform data format or API updates in a coordinated fashion. In order not to disrupt the behavior of the entire system (e.g., a sensor system controlling factory or greenhouse operations), deployments of software updates may have to be tiered and then delayed and synchronized in such as fashion that updates do not take effect until all the impacted devices are known to have received and processed the updates. Also, it should be kept in mind that many devices may have intermittent connectivity and may thus be unreachable or offline for extended periods of time; such devices may be unable to receive updates until much later. These kinds of situations pose challenges not only for software development but system operations as well.

IoT development is complicated further by novel software development approaches and assumptions that rely on rapid deployment cycles and small increments that are constantly taken to use. Continuous delivery and deployment technologies and DevOps methodologies [2], [11], [3] have redefined the expected behavior of software systems, resulting in updates that can potentially take place several times a day. Such an approach builds on an automated pipeline that starts from a development environment where programmers make changes, and ending in public deployment of the new system, with everything tested automatically along the way. The various phases of this pipeline include compilation, integration, testing, staging, deployment (potentially to a large number of independently running computing devices), and finally operations. Today, this way of working is well understood in the context of web-based online services. However, its full adoption in complex IoT systems can be difficult and more akin to challenges presented by *process automation systems* [12] than those challenges that are present in conventional PC or smartphone application deployment.

C. Other Important Emerging Trends and Predictions

Although in this paper our focus is on software development technologies and the overall IoT system architecture, we wish to briefly highlight important advances in communication technologies that will have a significant impact on the overall IoT system and software architecture.

Cellular IoT radio technologies will eliminate the need for gateways. In many ways, the presence of gateways in IoT systems can be viewed as a nuisance. Ideally, IoT devices should just work anywhere and out-of-the-box without any special installation or startup steps (such as setting up Bluetooth pairing or Wi-Fi security passwords). There are emerging Cellular IoT radio technologies such as NB-IoT

and LTE-M that will be deployed to existing cellular radio networks, providing nationwide coverage for IoT devices [13]. Ultimately, such low-power wide area network (LPWAN) technologies will eliminate or at least substantially reduce the need for gateways, enabling *direct communication between IoT devices and the cloud at reasonable cost and energy efficiency*.

Intelligence will increasingly move from the cloud towards the edge. Historically, IoT systems were very cloud-centric, with the majority of computation taking place centrally in the cloud. However, in recent years there has been a noticeable trend towards *edge computing*, i.e., systems in which the edge of the network (devices and gateways) plays a key role in filtering and processing the data. This will increase the importance of *mesh networking technologies* that allow IoT devices to perform actions and processing in a peer-to-peer (P2P) fashion with very low latencies. Together, the emergence of LPWAN and mesh networking technologies can be expected to significantly alter the topologies and the overall architecture of IoT systems.

Isomorphic IoT system architectures will emerge. Earlier in this paper, we argued that the software development technologies required for different parts of the IoT systems are very different. However, given the rapidly increasing computing and storage capacities of IoT devices, we foresee that within the next 5-10 years IoT devices will be able to host considerably more capable software stacks. Ultimately, this may lead us to *isomorphic IoT system architectures* in which the devices, gateways and the cloud will be able to run the same applications and services, allowing flexible migration of code between any element in the overall system.

Isomorphic architectures can be seen as the "holy grail" in IoT development. Instead of having to learn many different incompatible ways of software development, in an isomorphic architecture one base technology will suffice and will be able to cover all aspects of E2E development. At this point it is still difficult to predict which technologies will "rule them all", so to speak. Container-based architectures such as *Docker* (https://www.docker.com/) or *CoreOS rkt* (https://coreos.com/rkt/) seem like good guesses at this point, even though their memory and computing power requirements may seem exorbitant from the viewpoint of today's IoT devices. Amazon's *Greengrass* system (https://aws.amazon.com/greengrass/) also points out to a model in which the same development technology can be used both in the cloud and in IoT devices; in Greengrass, the programming platform is Amazon's Lambda.

Isomorphic IoT systems will dilute the roles of the cloud and the edge, leading us to "soup computing". Although fully isomorphic IoT systems are still years away, their arrival may ultimately dilute or even dissolve the boundaries between the cloud and its edge. Isomorphic systems will allow computations to be transferred dynamically and performed in those elements that provide the optimal performance, storage, network speed, latency and energy-efficiency characteristics, thus enabling the overall behavior of the IoT system to be optimized based on a "soup" of available, diverse computational elements in the overall end-to-end system.

V. Conclusions

The Internet of Things (IoT) represents the next significant step in the evolution of the Internet. We believe that this evolution will ultimately result in the creation of a *Programmable World* in which even the simplest things and most ordinary artifacts are connected to the Internet and can be controlled and programmed remotely. The possibility to connect, manage, configure and dynamically reprogram remote devices through local and global cloud environments will open up a huge variety of new use cases, services, applications and device categories, and enable entirely new product ecosystems.

In this paper we have taken a look at IoT development and argued that this area is much more complex than people tend to assume. While at the surface IoT development may not seem all that different from mobile or web development, in reality the development of end-to-end IoT systems requires an unusually broad spectrum of development technologies and skills. These skills cover nearly all aspects of modern software development, ranging from embedded software to web and mobile application development as well as cloud backend development, including analytics and machine learning. IoT system development is further complicated by the distributed nature of the end-to-end architecture as well as the general drive towards continuous delivery and deployments. Therefore, in summary, in its present form the development of end-to-end IoT systems is perhaps the most complex form of software development.

References

[1] Taivalsaari, A., Mikkonen, T.: Roadmap to the Programmable World: Software Challenges in the IoT Era. IEEE Software, Jan/Feb 2017 **34**(1) (2017) 72–80

[2] Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education (2010)

[3] Debois, P.: DevOps: A Software Revolution in the Making. Journal of Information Technology Management **24**(8) (2011) 3–39

[4] Weir, C., Noble, J.: Small Memory Software: Patterns for Systems with Limited Memory. Addison-Wesley (Software Pattern Series) (2000)

[5] VisionMobile: Developer Economics State of the Developer Nation, Q1 2016. http://www.visionmobile.com/reports/developer-economics-state-developer-nation-q3-2016 [Online; accessed 18-March-2017].

[6] VisionMobile: Cloud and Desktop Developer Landscape. http://www.visionmobile.com/product/cloud-and-desktop-developer-landscape/ [Online; accessed 5-March-2016].

[7] Postscapes: IoT Cloud Platform Landscape (2016) [Online; accessed 18-March-2017].

[8] Rotem-Gal-Oz, A.: Fallacies of Distributed Computing Explained. http://www.rgoarchitects.com/Files/fallacies.pdf [Online; accessed 20-April-2016].

[9] Deutsch, L.P.: Fallacies of Distributed Computing. White Paper [Online; accessed 28-October-2017].

[10] Laprie, J.C.: Dependable Computing: Concepts, Limits, Challenges. In: Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing, IEEE (1995) 42–54

[11] Fitzgerald, B., Stol, K.J.: Continuous Software Engineering: A Roadmap and Agenda. Journal of Systems and Software **123** (2017) 176–189

[12] Love, J.: Process Automation Handbook: A Guide to Theory and Practice. Springer-Verlag London (2007)

[13] Wang, Y.P.E., Lin, X., Adhikary, A., Grövlen, A., Sui, Y., Blankenship, Y., Bergman, J., Razaghi, H.S.: A Primer on 3GPP Narrowband Internet of Things. IEEE Communications Magazine **55**(3) (2017) 117–123